AADL Inspector 1.5

User Manual

Ellidiss Technologies
http://www.ellidiss.fr
aadl@ellidiss.fr

Contents

1 Introduction 4
2 Before starting 6
2.1 Installation 6
2.2 Distribution contents 6
2.2.1. Bin subdirectories 7
2.2.2. Config subdirectory 8
2.2.3. Examples subdirectory 10
2.2.4. Doc subdirectory 12
2.2.5. Command line options 12
2.3 License 13
3 Graphical Interface 15
3.1 Main menu and button bar 15
3.1.1. File menu 16
3.1.2. View menu 17
3.1.3. Wizards menu 17
3.1.4. Tools menu 29
3.1.5. Help menu 31
3.1.6. Button bar 32
3.2 Source files area 32
3.3 Processing tools area 35
3.3.1. Static Analysis 36
3.3.2. Schedulability analysis 36
3.4 Simulation area 39
3.4.1. Simulator action buttons 40
3.4.2. External I/O button 40
3.4.3. Simulation time lines 41
3.5 Status bar and Error Report 42

1 Introduction

AADL Inspector is an analysis framework for real-time critical and software intensive systems. The tool operates at a model level and does not require the final source code to be generated and executed. **AADL Inspector** can process the following kinds of architectures with appropriate abstractions:

- multi-threaded software (**RTOS**: Real-Time Operating System)
- multi-partitions software (**TSP**: Time and Space Partitioning)
- multi-processor distributed software with network communication

In order to be able to perform advanced model processing in a homogeneous way and to reduce the effort of developing new analysis plug-ins, **AADL Inspector** operates on a common language that can be either the original input or the result of a prior model transformation. The common language that has been chosen is the Architecture Analysis and Design Language (**AADL**) declarative model. The formal definition of the **AADL** language can be found in the **SAE AS-5506B** document and general information about this language is available on the **AADL** committee web site: www.aadl.info and the **AADL** resource pages www.openaadl.org.

AADL Inspector is packaged in a standalone distribution that minimizes installation and maintenance effort to ease the every day use of the product on standard personal computers or network servers. The product is available for both **Windows** and **Linux** platforms.

The goal of **AADL Inspector** is to encompass a variety of tools to process a complete **AADL** specification composed of a set of text files. These files can be created within **AADL Inspector** itself, loaded from pre-existing local or remote libraries or automatically generated by an import wizard. **AADL** files can also be organized into projects to facilitate the management of large models. The processing tools can be used to analyse various facets of the architecture or to offer code generation and documentation capabilities. These processing tools are organized in a modular and extendable way so that additional tools can easily be included.

The standard installation of **AADL Inspector** 1.5 implements the following verification rules:

- **AADL** syntactic analysis (*aadlrev*)
- deployed instance model overview and declarative model statistics (*Metrics*)
- architectural (static) semantics (*Naming*, *Legality*, *Consistency* and *ARINC653*)
- real-time (dynamic) semantics (**Cheddar** and **Marzhin**)

It is also possible to modify the current **AADL** specification either with a direct edition of the opened source text files, or by using a set of wizards that enable advanced functions such as:

- select the root of the system instance hierarchy
- set the thread priorities according to the given scheduling protocol (RM, DM)
- bind threads to available processors with allocation algorithms
- modify the main thread real-time properties in a spreadsheet
- apply uniform text formatting rules (autoformat)
- convert older versions of source text into **AADL** v2.1
- import **OMG UML MARTE** models into **AADL**

The current version of **AADL Inspector** covers the following standard definitions:

- **AADL** Core (v1.0, v2.0 and v2.1)
- **AADL** Behaviour Annex (v1.0 and revision draft)
- **AADL** Error Model Annex (v1.0 and 2.0)
- **AADL** Data Model Annex
- **AADL ARINC 653** Annex (v1.0 and revision draft)

The standard documents can be acquired online on the **SAE** website:
http://standards.sae.org/as5506b/
http://standards.sae.org/as5506/2/

2 Before starting

2.1 Installation

Installation of the product only requires the following easy actions:

- get a copy of the installation package for the desired platform (**Windows**, **Linux** 32 bits **or Linux** 64 bits) from the **Ellidiss** website: http://www.ellidiss.com/

- run the installation program on **Windows** or uncompress and expand the archive file on **Linux**.

- launch the `AADLInspector` executable file located in the appropriate `bin` subdirectory of the installation directory, or the corresponding desktop shortcut on **Windows**.

Downloaded packages usually come with a temporary trial license that can be used free of charge. If you purchased the product or this temporary license has expired, please contact **Ellidiss** customer support service to get the appropriate license information and installation procedure that fits your situation. A standard installation requires less than 40 Mbytes of free disk space.

2.2 Distribution contents

Once installed on the computer, the **AADL Inspector** installation directory contains the following subdirectories:

- `bin.w32`, `bin.l32` or `bin.l64` subdirectory
- `bin.common` subdirectory
- `config` subdirectory
- `examples` subdirectory
- `doc` subdirectory

`bin.w32` contains platform dependent executable files for **Windows** (32bit and 64bit), whereas `bin.l32` and `bin.l64` contain the corresponding files for **Linux** platforms.

`bin.common` contains `.jar` files that are common to all platforms.

Note that after a first launch of the tool, a directory is created to store temporary files. The actual location of this temporary directory can be customized by the `logDirectory` parameter in the `config/AIConfig.ini` file, or the `-l` command line option. The default location of the temporary directory is within the user's home directory.

2.2.1. Bin subdirectories

These directories contain the executable files for the current platform or **Java** archive files that are shared by all platforms. The only external requirement is the availability of a proper **Java** Run-time Environment (**JRE**) to run the simulator. These files are:

- `AADLInspector` main executable file
- `AIMonitor` remote process monitoring executable file
- `aadlrev` executable file (**AADL** syntactic analyser)
- `xmlrev` executable file (**XML** syntactic analyser)
- `sbprolog` executable file (**Prolog** engine)
- `cheddarkernel` executable file (**Cheddar** schedulability analyser)
- `Marzhin`, `VAgent` and `VCore` **Java** archive files for the **Marzhin** simulator

`aadlrev 2.7` is a standalone **AADL** syntactic analyser that is used by the **LMP** (Logic Model Process) plug-ins to convert **AADL** specifications into a list of **Prolog** predicates. This utility tool can analyse textual **AADL** files that comply with **AADL** 2.1 (*SAE AS-5506B*), the **AADL** Error Model v1 (*SAE AS-5506/1 Annex E*), the **AADL** Behaviour Annex (*SAE AS-5506/2 Annex D*), and the **AADL** ARINC 653 Annex (*SAE AS-5506/2 Annex F*). In addition, the last draft of the new **AADL** Error Model v2 (future *SAE AS-5506/3 Annex E*) is also supported by `aadlrev`. Most of the **AADL** 1.0 (*SAE AS-5506*) and 2.0 (*SAE AS-5506A*) syntax is also recognized and can be automatically converted into the newest 2.1 format.

`xmlrev 1.1` is a standalone **XML** syntactic analyser that is used by the **LMP** (Logic Model Process) plud-ins to convert **XML** or **XMI** serialized models into a list of **Prolog** predicates.

`cheddarkernel 3.0.2` is a command-line version of the **Cheddar** v3 schedulability analysis tool. **Cheddar** v3 models (`.xmlv3`) are generated from the **AADL** specification

thanks to a dedicated **LMP** model transformation. **Cheddar** outputs (feasibility test reports and static time lines) are displayed in the **AADL Inspector** graphical interface. **Cheddar** is an open source project managed by the University of Brest: http://beru.univ-brest.fr/~singhoff/cheddar

`sbprolog 2.5` is an open source **Prolog** engine that is used by the **LMP** (Logic Model Processing) technology. **AADL Inspector** uses **LMP** to implement the various **AADL** rules checkers and model transformations. **SB-Prolog** was developed by State University of New York at Stony Brook and the University of Arizona.

`marzhin 2.2` is a multi-agent simulator implementing the **AADL** run-time. It consists of three **Java** archive files and requires a **Java** Run-time Environment (**JRE**) to operate. No **JRE** is provided with the **AADL Inspector** distribution. **Marzhin** v2 models (`.xml`) are generated from the **AADL** specification thanks to a dedicated **LMP** model transformation. **Marzhin** outputs (dynamic time lines) are displayed in the **AADL Inspector** graphical interface. **Marzhin** is developed in collaboration by **Virtualys** and **Ellidiss** Technologies.

2.2.2. Config subdirectory

This directory contains initialization and configuration files that are used by the executable files.

The files having a `.sbp` extension contain a binary form of the **LMP** (Logic Model Processing) rules that are used to perform each analysis action. Checkers provide a direct textual report into the **AADL Inspector** window, whereas adaptors perform dedicated model transformations to interface with ancillary tools such as **Cheddar** and **Marzhin**. More rules files will be added in the future and it is also possible to include corporate or project specific checking or adaptor tools by adding a rule file into this directory ("plug and check" feature). Please contact **Ellidiss Technologies** (aadl@ellidiss.fr) to use this advanced feature.

The files having a `.ais` extension contain a description of each **AADL Inspector** processing tool or wizard. Each processing tool or wizard provides one or more services that will be available via menu options or buttons. Each service is described by a sequence of elementary commands.

The `AIConfig.ini` file contains the declaration of three sets of user variables: *config*, *plugins*, *gantt*, *accelerators* and *license*. These options are not supposed to be changed by the end user without assistance from technical support.

With the standard distribution, the `config` directory contains the following additional sub-directories and files:

2.2.2.1. Common plugins

These plugins can be removed and customized. New plugins can also be added here. They are not platform dependent. They are located in the `plugins.common` subdirectory.

AADL Static Analysis:

- `AadlAnalysis.ais`: tool description file
- `metrics.sbp`: **AADL** metrics checker
- `naming.sbp`: **AADL** naming rules checker
- `legality.sbp`: **AADL** legality rules checker
- `consistency.sbp`: **AADL** consistency rules checker
- `arinc653.sbp`: **ARINC 653** rules checker

Schedulability Analysis:

- `Cheddar.ais`: tool description file
- `schedulability.sbp`: **AADL** to **Cheddar** model transformation

Simulation:

- `chronogram.sbp`: time lines configuration rules
- `marzhin.sbp`: **AADL** transformation rules for **Marzhin**
- `Marzhin.xml`, `MarzhinLogs.xml`: simulation configuration files

Wizards:

- `aadlgen.sbp` : **AADL** generator (unparser)
- `readRTProperties.sbp` : **AADL** real-time properties reader
- `writeRTProperties.sbp` : **AADL** real-time properties writer
- `rootselector.sbp`: **AADL** instance model configuration rules
- `Marte.ais`: wizard description file
- `marte.sbp`: **UML MARTE** to **AADL** transformation rules

Scripts:

- `aiscripts.ais`: scripts encapsulation plugin description file (refer to §2.2.2.3 for further details)

2.2.2.2. Platform dependent plugins

The subdirectories `plugins.w32 and plugins.l32` are not currently used. Their role is to give the ability to call platform dependent commands during the execution of a service provided by a processing tool or wizard.

2.2.2.3. AADL Inspector scripts

The script subdirectory contains a few examples of command files that can be used to define sequences of actions to be executed automatically via the *AI Scripts* plugin or the command line. The number of available commands is currently restricted but will be extended in the next versions of the tool in order to provide a fully scriptable **API** (Application Programming Interface) for **AADL Inspector**.

- `compareDataflow.script`: opens the `examples/dataflows.aic` project, then runs **Cheddar** to compute the *Schedule Table* on the current project and save the results in a file in **VCD** format (Value Change Dump), then runs **Marzhin** to simulate the current project and save the results in a file in **VCD** format, and finally compares the two **VCD** traces.
- `compareSharedData.script`: runs the same sequence of actions for the `examples/shared_data.aic` project.
- `compareARINC653.script`: runs the same sequence of actions for the `examples/ arinc653.aic` project.
- `compareAllProjects.script`: runs the same sequence of actions for all the projects defined in the `examples` directory. Note that running this script may take a significant time.

2.2.3. Examples subdirectory

This directory contains a few **AADL** examples to practice the use of **AADL Inspector**. Two kinds of files are proposed: individual **AADL** source files (`.aadl`) or **AADL** Inspector project files (`.aic`) that provide a list of individual **AADL** source file pathnames or **URL**s. It is recommended that a project file is loaded rather than individual

AADL files, to ensure all the required **AADL** Packages and Property Sets that are required to activate the analysis tools are opened.

- `dataflow.aic`: dataflow communication between threads. It can be used to observe the effect of sampled, immediate and delayed data port connections.
- `messages.aic`: message based communication between threads using queued events. It can be used to observe input queue overflow.
- `shared_data.aic`: shared data communication between threads. It can be used to observe the effect of the *Priority_Ceiling_Protocol* to avoid a deadlock.
- `client_server.aic` : subprogram call communication between threads. It can be used to observe the effect of the client-server synchronisation protocols.
- `dispatching.aic`: various thread dispatching protocols. It can be used to compare the behaviour of *Periodic*, *Sporadic*, *Aperiodic*, *Hybrid*, *Timed* and *Background* threads.
- `display_system.aic`: a large model (5 processors, 22 processes and 123 threads). It can be used to check the scalability of the tools. Note that the analysis of this model can take a few minutes.
- `mars_pathfinder.aic`: several threads with different priority sharing common data. It can be used to observe the priority inversion problem.
- `satellite.aic`: a model defined in the **AADLib** github. It can be used to experiment remote model loading via the internet.
- `radar.aic`: a simplistic radar system
- `arinc653.aic`: two layer hierarchical scheduling. It can be used to investigate time and space partitioned systems.
- `canbus.aic`: bus communication between processors. It can be used to observe the impact of bus latency on thread scheduling.
- `multicore.aic`: partitioned scheduling on a dual-core processor. It can be used to practice the automatic thread placement wizard.
- `calculator.aic`: integer arithmetic with the **AADL Behaviour Annex**. It can be used to show the math library capabilities and the interaction between the user and the simulator.
- `coffee.aic`: a coffee machine control system. It can be used to show conditional computation with the **AADL Behaviour Annex**.
- `regulator.aic`: a temperature regulation system. It can be used to illustrate the design and analysis of a discrete control system with the **AADL Behaviour Annex**

In addition, a few examples of **UML MARTE** models are provided. They are showing different variants in the use of this standard. In all cases, the **XMI** model contained in the selected `.uml` file is parsed and transformed into an **AADL** model that is automatically loaded in **AADL Inspector** for further analysis. These models can be opened through the *import UML MARTE* wizard.

2.2.4. Doc subdirectory

This directory contains a list of `.pdf` files (including this manual) that can be opened from the help menu of the graphical user interface.

2.2.5. Command line options

AADL Inspector can be launched from a command line. The following optional parameters are available:

- `-a file1.aadl, file2.aadl,…`
  open the specified **AADL** files
- `-r dir1, dir2,…`
  open all the **AADL** files contained in the specified directories
- `-p project1.aic, project2.aic,…`
  open all the **AADL** projects contained in the specified directories.
- `-l logdirname`
  use the specified location to create the temporary files. If used, this information overrides the one specified by the *logDirectory* parameter in the `AIConfig.ini` file.
- `--plugin tool.service`
  start a service of a tool as defined in a `.ais` file of the config directory.
- `--result file`
  store the result file in the specified file
- `--result stdout`
  only for **Unix** systems. Display the plugin result in the console.
- `--show false`
  launch **AADL Inspector** without showing the graphical interface.

An example of **AADL Inspector** being launched from the command line is its integrated use with the **Stood AADL** graphical modelling tool. With recent versions of **Stood**

(**Stood** 5.3 or more recent), **AADL Inspector** is automatically called after the generation of textual **AADL** code has been completed. The list of **AADL** files that are loaded are all those related to the **Stood** project, i.e. the current design as well as the various libraries.

Another example of use of the command line activation of **AADL Inspector** is to run **Cheddar** on a set of specified **AADL** files and get the results in a specified output file:

```
bin.l32/AADLInspector
   –p examples/dataflow.aic
   --plugin Schedulability.cheddarTheoTest
   --result dataflow.xml
   --show false
```

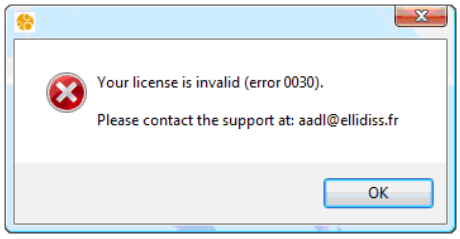
Such a command will create a file containing the result below (fragment). The detailed description of the **Cheddar** output is provided in a separate annex document.

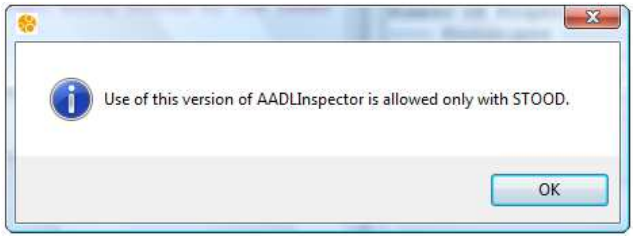
```
<results>
  <feasibilityTest name="processor utilization factor" …>
    <computation name="base period" reference="all" value="300" …/>
    <computation name="processor utilization factor with deadline"
     reference="all" value="0.78333" …/>
    <computation name="processor utilization factor with period"
     reference="all" value="0.78333" …/>
    …
  </feasibilityTest>
  <feasibilityTest name="worst case task response time" …>
    <computation name="response time"
     reference="root.my_platform.CPU.my_process.T1" value="15" …/>
    <computation name="response time"
     reference="root.my_platform.CPU.my_process.T2" value="10" …/>
    <computation name="response time"
     reference="root.my_platform.CPU.my_process.T3" value="5" …/>
    …
  </feasibilityTest>
</results>
```

2.3 License

A valid license is required to use **AADL Inspector**. Various kind of licenses are available, including free of charge evaluation and education licenses. Payment of a license fee is required for commercial or industrial usage of **AADL Inspector**. Please contact your **Ellidiss** sales representative for more details.

License information is stored in the `AIConfig.ini` file. Licenses can be attached to a particular computer and limited in time. In case of a mismatch between the license information stored in the `AIConfig.ini` file and the computer identification or the current date, an error message box is displayed.



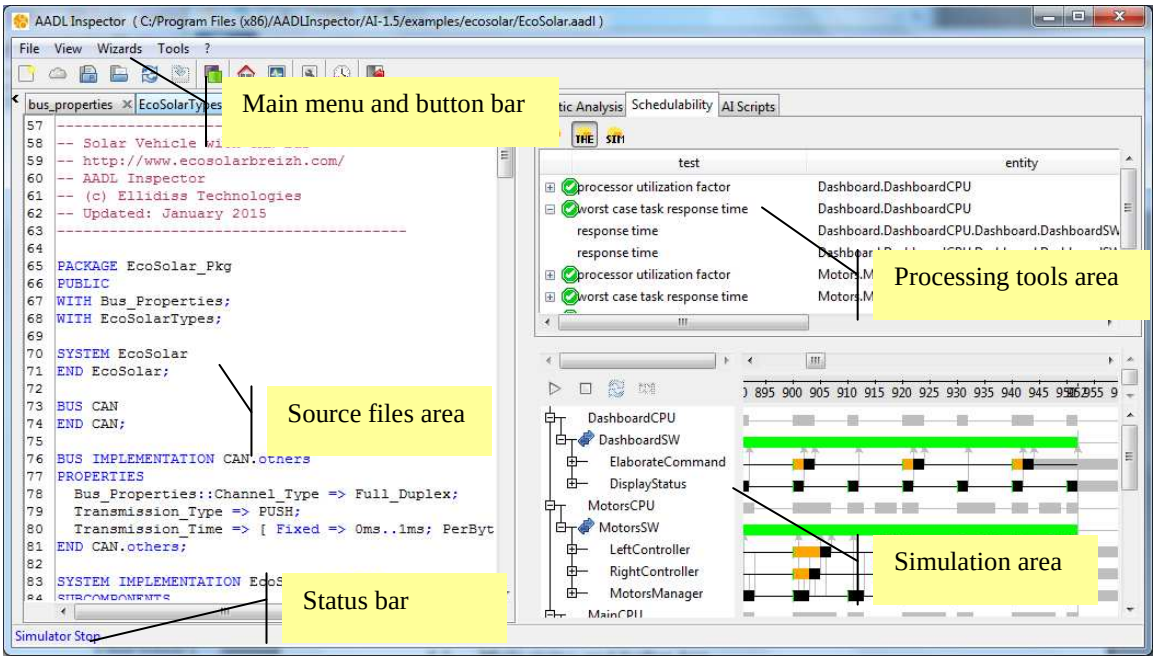


An error number is provided to help identify the license problem:

- `0010`: this license has expired
- `0020`: this license is not compatible with the encryption key
- `0030`: this license is attached to another computer
- `0040`: this license is linked to a **Stood** license
- `0050`: this license is not valid for this version of the product
- `0060`: this licence is invalid (empty licenseKey)
- `0070`: this licence is invalid (unspecified version)
- `0080`: this licence is invalid (bad licenseKey encoding)
- `0090`: this licence is invalid (bad version encoding)

3 Graphical Interface

AADL Inspector opens a single window that encompasses a main menu bar, a button bar, a source files area, a processing tools area, a simulation area and a status bar, as shown below:

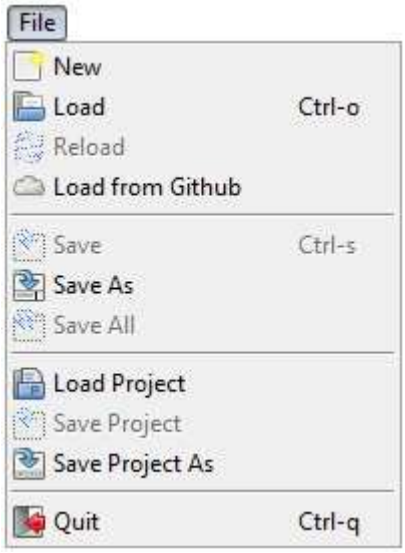


3.1 Main menu and button bar

The menu bar contains the following pull-down menus: *File*, *View*, *Wizards*, *Tools* and *Help*. The button bar provides shortcuts for frequently used menu options.

3.1.1. File menu

The *File* menu controls all actions related to the **AADL** files that are processed by the tool. The tool can process several files that together define a complete **AADL** specification. The recommended way to manage multiple files is to link them with an **AADL Inspector** project file (`.aic`). There is no particular restriction or naming rule for the **AADL** files. In practice, for most of the processing tools, all the loaded files are concatenated together before being processed by the analysis tools. Please note that load ordering may have an impact on obtained result, especially if the root of the **AADL** instance hierarchy has not been explicitly defined.

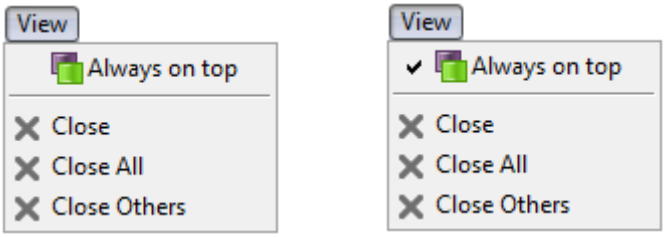


- *New*: create a new **AADL** virtual file in memory.
- *Load*: load the contents of the specified **AADL** files into memory.
- *Reload*: cancel any non saved change for the current file.
- *Load from Github*: load files from remote **AADL** libraries (requires internet access).
- *Save*: save the changes for the current file.
- *Save As*: save the current memory contents into a new file.
- *Save All*: save the changes for all the loaded files.
- *Load Project*: load the content of all the **AADL** files of the specified `.aic` project into memory.
- *Save Project*: save the changes for the current project.

- *Save Project As*: save all the opened files into a new `.aic` project.
- *Quit*: quit **AADL Inspector**

3.1.2. View menu

The *View* menu provides an option that constrains the **AADL Inspector** window to remain in the foreground of the computer screen.

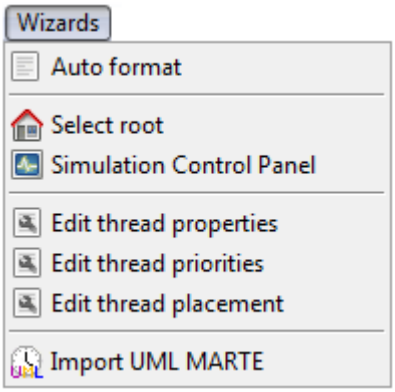


Once this menu item has been selected, a tick is put against its label to indicate that this constraint has been enabled. Another activation of this option disables the constraint and removes the tick.

The other items displayed in this menu are:
- *Close*: unload the current file.
- *Close All*: unload all the loaded files.
- *Close Others*: unload all the loaded files but the current one.

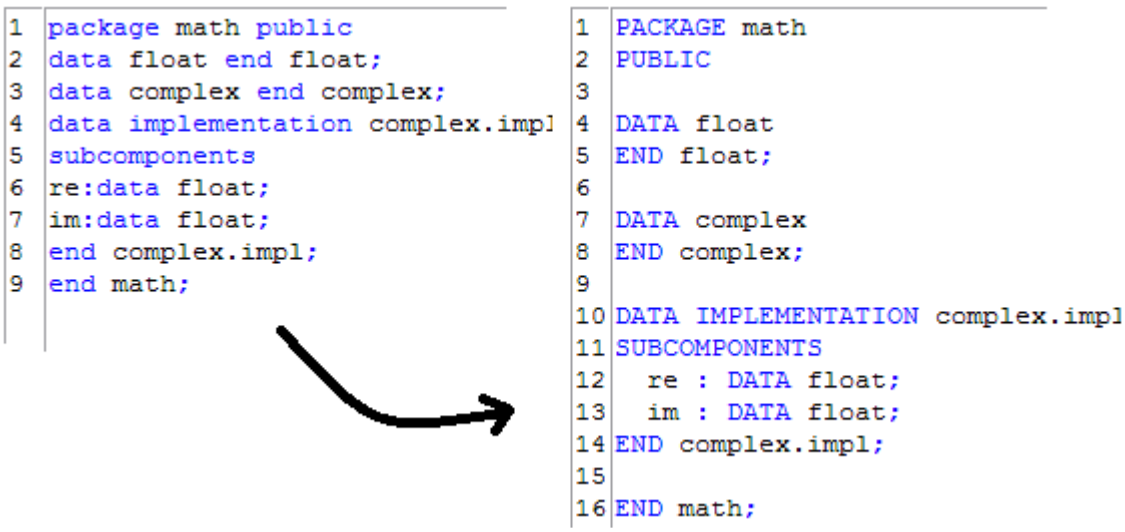
3.1.3. Wizards menu

The *Wizards* menu provides advanced functions used to perform changes on the input **AADL** specification. When possible, the original source text is not modified, and the changes are applied to an extension of the main system implementation of the project instead.



3.1.3.1. Auto format

This wizard re-writes the current **AADL** file into a normalized form. It impacts the case of identifiers and keywords, the indentation and the number of blank lines. This feature can also be used to convert **AADL** v1 and v2.0 files into **AADL** 2.1 syntax, except for some values of Property Associations.



The *Auto format* wizard runs the **AADL** parser on an original **AADL** specification as shown on the left hand side of the picture above, performs an « identity » model transformation and then applies the **AADL** unparser to get a formatted **AADL** specification as shown on the right hand side.

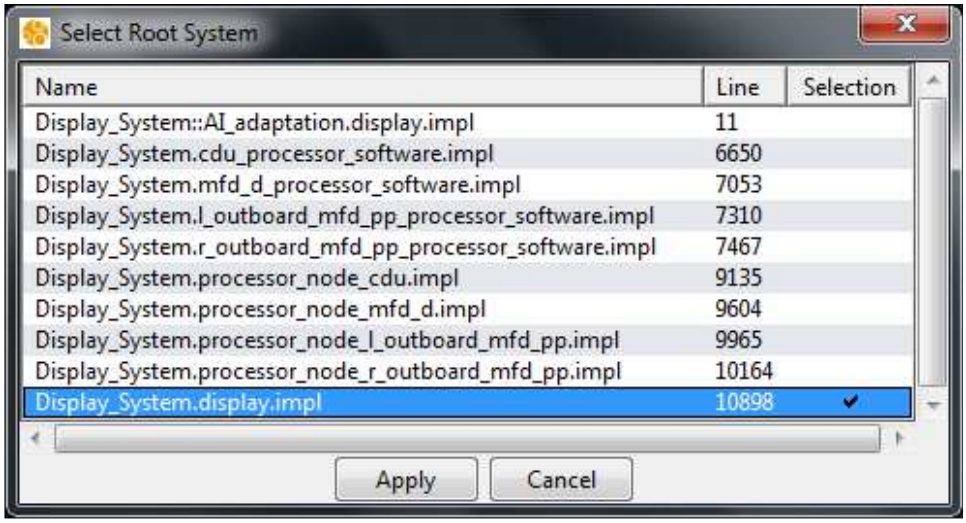
Note that it is possible to customize the format produced by the *Auto format* wizard thanks to dedicated **AADL** properties. These properties can be applied to any **AADL** entity, but we recommend inserting them at the Package level. The currently supported **AADL** unparser properties control the case of identifiers and keywords, as well as the automatic insertion of a header.

```
PROPERTY SET lmp IS

  unparser_id_case: ENUMERATION (AsIs,Upper,Lower) => Lower
  APPLIES TO (ALL);

  unparser_kw_case: ENUMERATION (AsIs,Upper,Lower) => Upper
  APPLIES TO (ALL);

  unparser_insert_header: ENUMERATION (Yes,No) => No
  APPLIES TO (ALL);

END lmp;
```

The next picture shows an example of use of these formatting properties.

```
1  ----------------------------
2  -- AADL-2.1
3  -- aadlrev2.7
4  -- (c)Ellidiss Technologies
5  -- 30Jan2015
6  ----------------------------
7
8  package MATH
9  public
10
11 data FLOAT
12 end FLOAT;
13
14 data COMPLEX
15 end COMPLEX;
16
17 data implementation COMPLEX.IMPL
18 subcomponents
19   RE : data FLOAT;
20   IM : data FLOAT;
21 end COMPLEX.IMPL;
22
23 properties
24   lmp::unparser_id_case => upper;
25   lmp::unparser_kw_case => lower;
26   lmp::unparser_insert_header => yes;
27 end MATH;
28
```

3.1.3.2. Select root



Most of the analysis and processing tools require the **AADL** declarative model to be instantiated and deployed first. **AADL Inspector** does not require this instantiation to be done statically, and the **AADL** instance model is not stored to avoid the risk of processing an outdated model. In practice, the instance model is built on the fly together with the proper model transformation that is required for each processing tool.

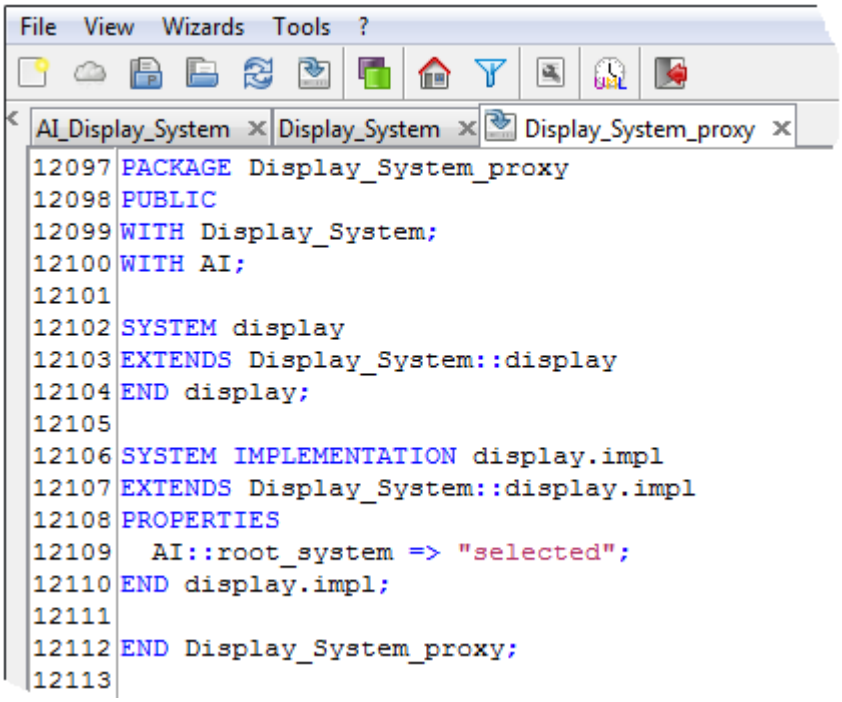
However, several instance models can be inferred from a given declarative model. It is thus mandatory to define which System Implementation represents the root of the instance hierarchy (System Instance). The *Select root* wizard provides the list of candidate System Implementations and selects the one to be the root of the **AADL** instance hierarchy.

The root system that will be considered by the analysis tools will be (in decreasing priority order):

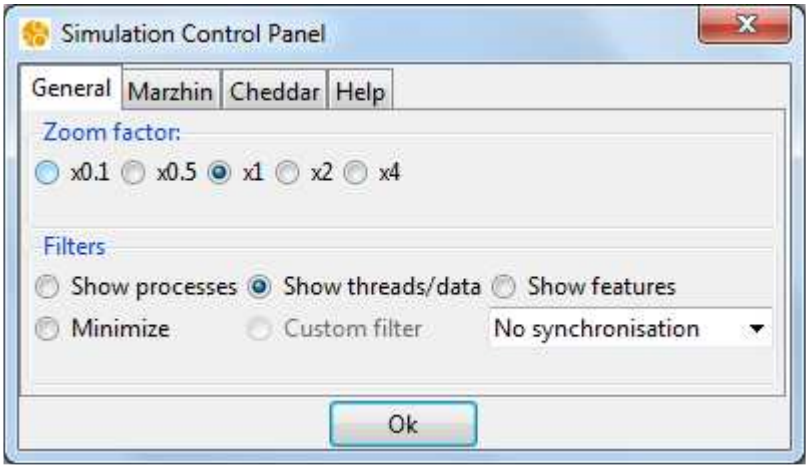
- the first found System Implementation containing an `AI::Root_System` Property association with the value `"SELECTED";`
- the first found System Implementation containing an `AI::Root_System` Property association with any other value;
- the first found System Implementation containing an `Actual_Connection_Binding` Property association;

- the first found System Implementation containing an `Actual_Processor_Binding` Property association;
- the first found System Implementation containing an `Allowed_Processor_Binding` Property association;
- the first found System Implementation that is not instantiated as a Subcomponent in the scope of the current Project.

If another root is selected by the user, an extended root system is created in memory only and is located in a new proxy package. The newly created system contains an `AI::Root_System => "SELECTED"` property association so that it becomes the new current root system. Note that if the user wants to use this property to define the root system in the original file, the property value `"SELECTED"` must not be used as it is reserved for the proxies.



3.1.3.3. Simulation Contol Panel



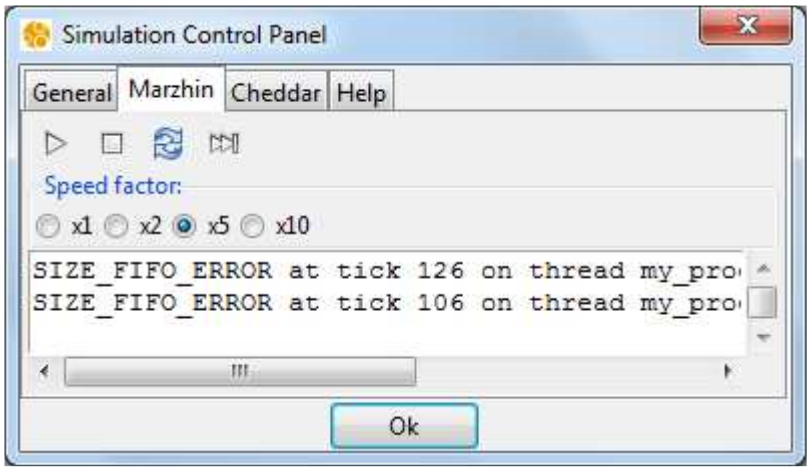
This wizard can be opened from the main menu or button bar and is also automatically opened when the Marzhin simulator is started. It is composed of four tabs that can be used to control the display and behaviour of the time simulators.

The *General* tab controls the appearance of the time lines frame. The horizontal axis (time) can be squeezed of extended with the *Zoom factor*. Note that the zoom factors can be customized in the `AIConfig.ini` configuration file. The vertical axis (model entities) can be selectively deployed thanks to the display *Filters*. The effect of these filters is described below:

- *Minimize*: only displays the Processors and the Bus, and thus reduces the drawing time.
- *Show processes*: adds a time line for each Process.
- *Show threads/data*: adds a time line for each Thread and shared Data subcomponent.
- *Show features*: adds a time line for each ports and data access feature.
- *Custom filter*: this option is selected when the display filters are directly controlled from within the simulation display area.

The selected filter applies to both the Cheddar Schedule Table and the Marzhin simulation trace. However, when a custom filter is set, it is possible to decide if the changes done on one simulation frame will no be mirrored on the other one (*No*

synchronization), or it will be propagated to the other one (*Sync on Marzhin* or *Sync on Cheddar ST*).



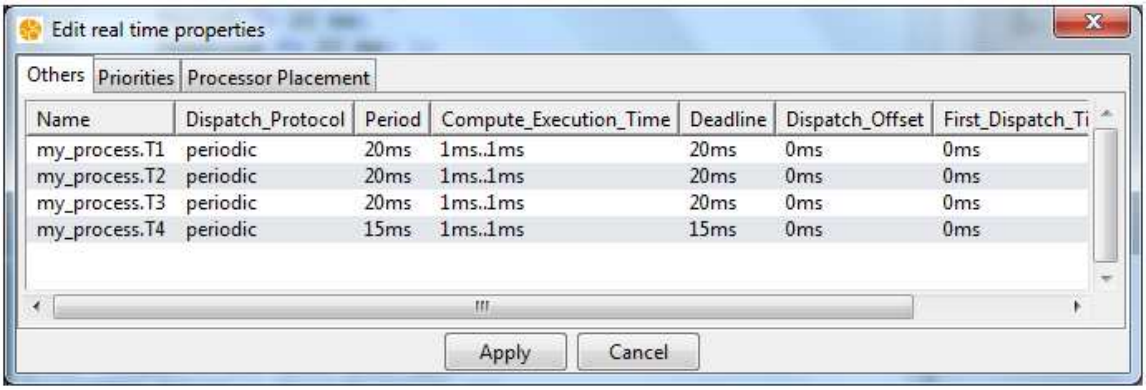
The *Marzhin* tab is used to interact with the **Marzhin** simulator. It contains a remote panel of the simulator main commands (*start/pause/resume*, *stop*, *refresh* and *go to last tick*) that are described in section 3.4.1, a *Speed factor* selector and a display area where message issued by the simulator are shown.

The *Cheddar* tab can be used to define the time window for computing the **Cheddar** static simulation (*Cheddar Schedule Table*). Minimizing this range can significantly reduce the computation time on large models.

The *Help* tab provides a caption for the colour code that is associated with the various states of the modelling entities that are observed during the simulation. The default values are explained in section 3.4.2. Note that this colour code can be customized in the `AIConfig.ini` configuration file.

3.1.3.4. Edit thread properties

This wizard opens a spreadsheet to edit various real-time Properties and apply them to the current model.



When real-time properties have been modified, the corresponding **AADL** Property associations are declared as contained Properties of an extension of the current root System Implementation. The extended root System is created in memory only and is

located in a new proxy Package. The newly created System contains an `AI::Root_System` Property association so that it becomes the new current root System to ensure that the new Property values are used.

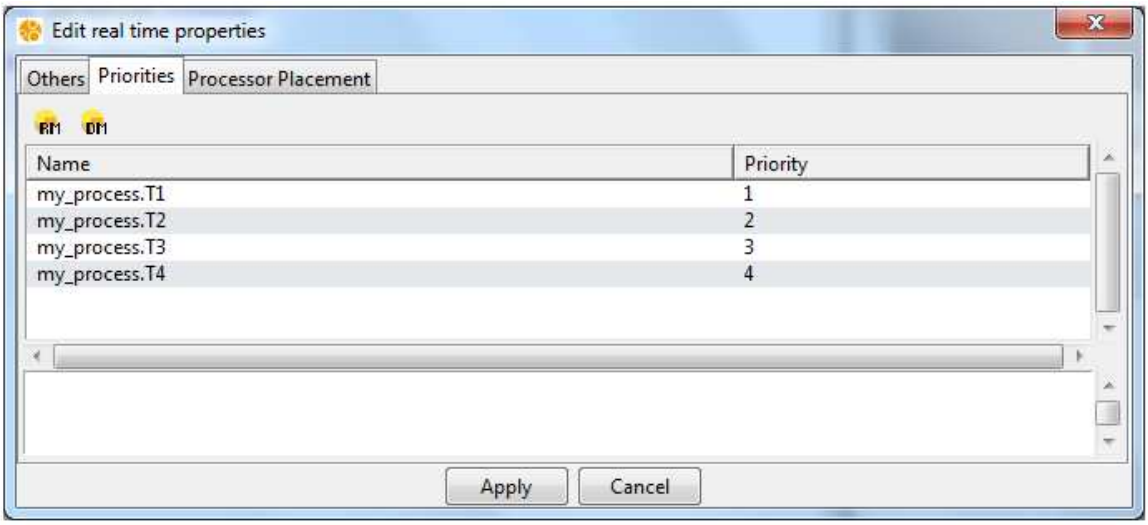
Base_Types | math | HW | synchronous | dataflow_Pkg_proxy

```
542 PACKAGE dataflow_Pkg_proxy
543 PUBLIC
544 WITH dataflow_Pkg;
545 WITH AI;
546
547 SYSTEM dataflow
548 EXTENDS dataflow_Pkg::dataflow
549 END dataflow;
550
551 SYSTEM IMPLEMENTATION dataflow.others
552 EXTENDS dataflow_Pkg::dataflow.others
553 PROPERTIES
554   AI::root_system => "SELECTED";
555   Dispatch_Offset => 5ms APPLIES TO my_process.T3;
556 END dataflow.others;
557
558 END dataflow_Pkg_proxy;
559
```

Note that **AADL** time unit conversion is not taken into account. To obtain consistent results, the same time unit should be used for all the timing properties in the **AADL** project.

3.1.3.5. Edit thread priorities

This wizard opens a spreadsheet to manually specify or automatically compute the Threads priority according to rate monotonic (**RM**) or deadline monotonic (**DM**) algorithms.



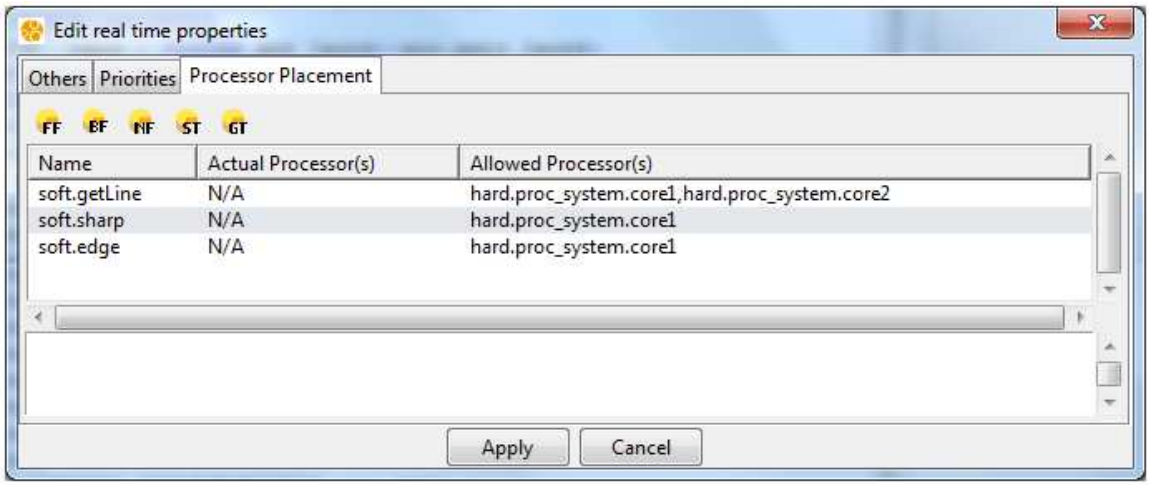
When priorities have been modified, the corresponding **AADL** Property associations are declared as contained Properties of an extension of the current root System Implementation. The extended root System is created in memory only and is located in a new proxy Package.

3.1.3.6. Edit thread placement

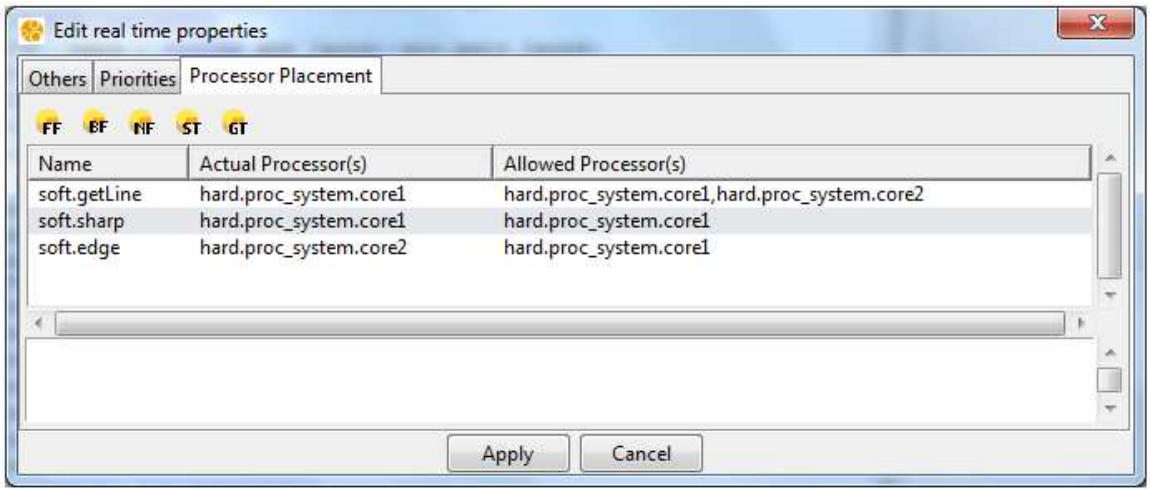
This wizard opens a spreadsheet to automatically compute the Threads placement onto the available Processors according to various placement algorithms.

```
574 SYSTEM IMPLEMENTATION product.impl
575 SUBCOMPONENTS
576   hard : SYSTEM soc_leon4::soc.asic_leon4;
577   soft : PROCESS edgeDetection.impl;
578 PROPERTIES
579   allowed_processor_binding => (
580     REFERENCE(hard.Proc_System.Core1),
581     REFERENCE(hard.Proc_System.Core2) ) APPLIES TO soft.getLine;
582   allowed_processor_binding => (
583     REFERENCE(hard.Proc_System.Core1) ) APPLIES TO soft.sharp;
584   allowed_processor_binding => (
585     REFERENCE(hard.Proc_System.Core1) ) APPLIES TO soft.edge;
586 END product.impl;
```

As shown above, the original model must contain a set of Threads located in a global Process that is bound to a group of Processors with `Allowed_Processor_Binding` Property associations. This initial situation is reflected in the Processor Placement wizard. As follows:



Then, it is possible to apply one of the placement algorithms that are proposed by **Cheddar**: first fit (*FF*); best fit (*BF*); next fit (*NF*); small task (*ST*) or general task (*GT*).



When the proposed placement is accepted (*Apply* button), the wizard generates **AADL** `Actual_Processor_Binding` Property associations in an extension of the current

root System Implementation. The extended root System is created in memory only and is located in a new proxy Package, as shown in the code fragment below. This System becomes the new default root, so that the analysis tools can work on the distributed Threads.

soc_leon4 | Processor_Bus_System | app | app_proxy | app_proxy_placement

```
643 PACKAGE app_proxy
644 PUBLIC
645 WITH app;
646 WITH AI;
647 WITH app_proxy_placement;
648
649 SYSTEM product
650 EXTENDS app::product
651 END product;
652
653 SYSTEM IMPLEMENTATION product.impl
654 EXTENDS app::product.impl
655 SUBCOMPONENTS
656   ea_core1 : PROCESS app_proxy_placement::ea_core1.impl;
657   ea_core2 : PROCESS app_proxy_placement::ea_core2.impl;
658 PROPERTIES
659   AI::root_system => "SELECTED";
660   Actual_Processor_Binding => REFERENCE(hard.proc_system.core1)
661     APPLIES TO ea_core1;
662   Actual_Processor_Binding => REFERENCE(hard.proc_system.core2)
663     APPLIES TO ea_core2;
664 END product.impl;
665
666 END app_proxy;
667
```

Another proxy_placement Package is also created in memory to declare a new Process per Processor, which encompasses the group of Threads that have been allocated to this Processor. An example of the corresponding **AADL** code is shown below:

soc_leon4 | Processor_Bus_System | app | app_proxy | app_proxy_placement

```
666 PACKAGE app_proxy_placement
667 PUBLIC
668 WITH app;
669
670 PROCESS ea_core1
671 END ea_core1;
672
673 PROCESS IMPLEMENTATION ea_core1.impl
674 SUBCOMPONENTS
675   getLine :  THREAD app::getLine.impl;
676   sharp :  THREAD app::sharp.impl;
677 END ea_core1.impl;
678
679 PROCESS ea_core2
680 END ea_core2;
681
682 PROCESS IMPLEMENTATION ea_core2.impl
683 SUBCOMPONENTS
684   edge :  THREAD app::edge.impl;
685 END ea_core2.impl;
686
687 END app_proxy_placement;
688
```

3.1.3.7. Import UML MARTE

This wizard loads a `.uml` file in **XMI** syntax, applies the **MARTE** to **AADL** model transformation and displays the generated **AADL** specification. Note that there are many ways to describe real-time systems with the **UML MARTE** profile.

A few examples of `.uml` files are provided in the `examples/MARTE` directory.

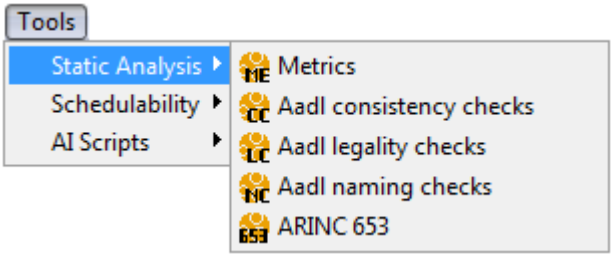
Note that the current transformation rules already take into accounts several variants and will be improved according to feedback received from users.

3.1.4. Tools menu

The *Tools* menu provides access to the processing tools and services that are defined in the `.ais` files located in the `config` directory. Three tools are available with the

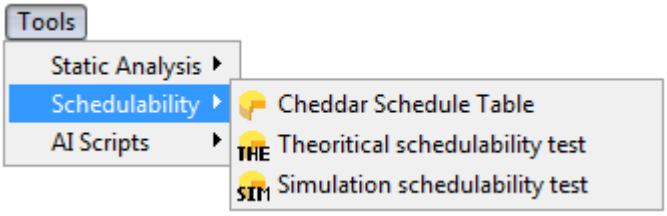
standard distribution: *Static Analysis*, *Schedulability* and *AI Scripts*. The second one uses the **Cheddar** tool.

3.1.4.1. Static Analysis



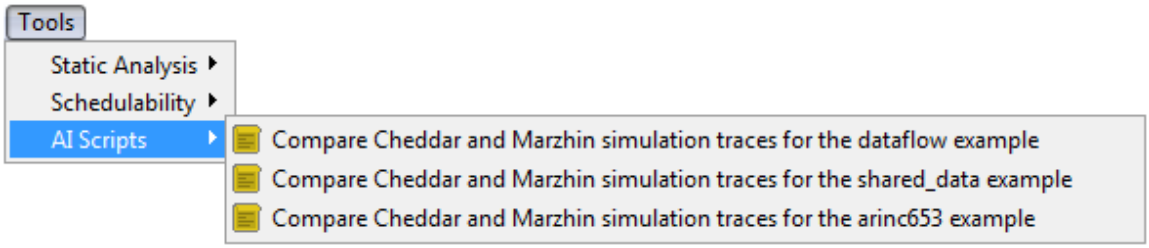
- *Metrics*: provide statistics about both the instance and the declarative **AADL** models.
- *AADL consistency checks*: verify the consistency rules defined by the standard.
- *AADL legality checks*: verify the consistency rules defined by the standard.
- *AADL naming checks*: verify the consistency rules defined by the standard.
- *ARINC 653*: set of rules for partitioned systems.

3.1.4.2. Schedulability Analysis



- *Cheddar Schedule Table*: static simulation computed by **Cheddar**.
- *Theroritical schedulability test*: set of feasibility tests checked by **Cheddar**.
- *Simulation schedulability test*: set of tests based on the static simulation computed by **Cheddar**.

3.1.4.3. Automated Scripts

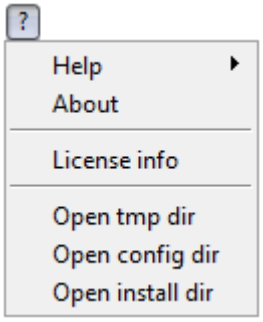

Tools
Static Analysis
Schedulability
AI Scripts
Compare Cheddar and Marzhin simulation traces for the dataflow example
Compare Cheddar and Marzhin simulation traces for the shared_data example
Compare Cheddar and Marzhin simulation traces for the arinc653 example

It is possible to create scripts to execute pre-defined sequences of actions. Only a restricted set of commands can be used currently in theses scripts. More scriptable commands will be added in the future versions of **AADL Inspector**.

The three examples that are proposed are used to automate the comparison of the time lines generated by the static simulation computed by **Cheddar** and the ones produced by the dynamic simulation of **Marzhin**. The two simulation traces are stored in **VCD** format so that they can be compared.

3.1.5. Help menu

The *?* menu provides information about **AADL Inspector**.

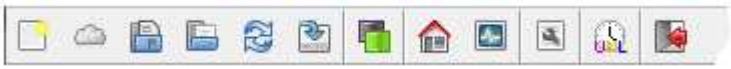


- *Help*: open the help files. Note that the name of the help file directory and the application that is used to open it can be customized in the `AIConfig.ini` file. By default, this application will be the default one for `.pdf` files on **Windows** and `xpdf` on **Linux**.
- *About*: display the version of the software.

- *License info*: provide information about the license.
- *Open tmp dir*: open the temporary subdirectory.
- *Open config dir*: open the configuration subdirectory.
- *Open install dir*: open the installation subdirectory.

3.1.6. Button bar

The button bar provides another entry point for selected menu actions.



The effect of these actions is described in the corresponding menu section. Button association with menu bar items is given below from left to right:

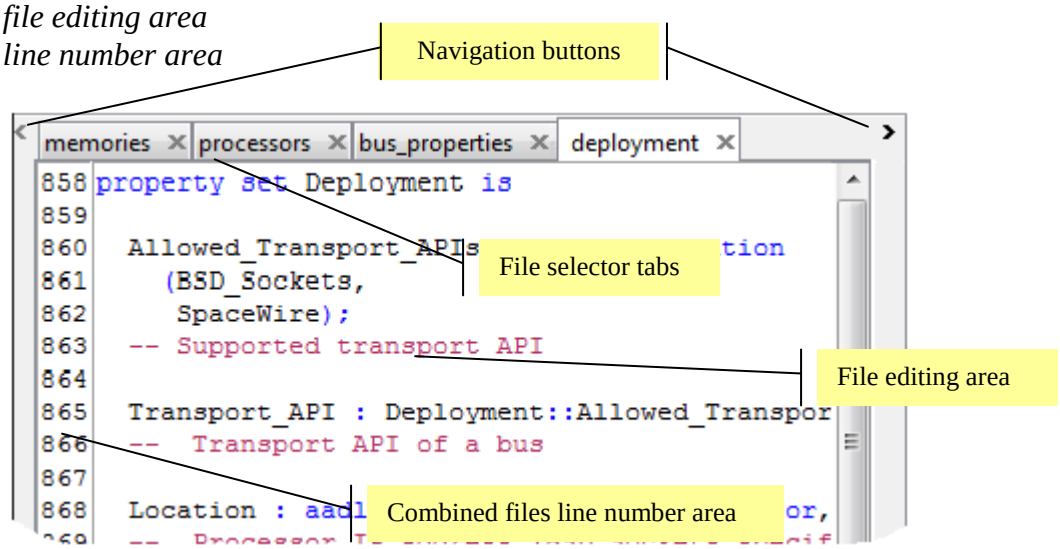
- *File/New*
- *File/Load from Github*
- *File/Load Project*
- *File/Load*
- *File/Reload*
- *File/Save*
- *View/Always on top* (toggle button)
- *Wizards/Select root*
- *Wizards/Configure simulations*
- *Wizards/Edit thread properties, priorities and placement*
- *Wizards/Import UML MARTE*
- *File/Quit*

3.2 Source files area

The **AADL** files that are processed by the tool are managed in the *Source files area*. The tool can process several files that together define a complete **AADL** specification. There is no particular restriction or naming rule for the **AADL** files. An **AADL** file can contain one or more **AADL** Packages or Property Sets. In practice, all the loaded files are concatenated before being processed by the analysis tools. Please note that load ordering may have an impact on obtained result, especially while looking for the root of the **AADL** instance hierarchy. Displayed line numbers reflect this ordering as it refers to the resulting file concatenation.

The source file area is composed of:

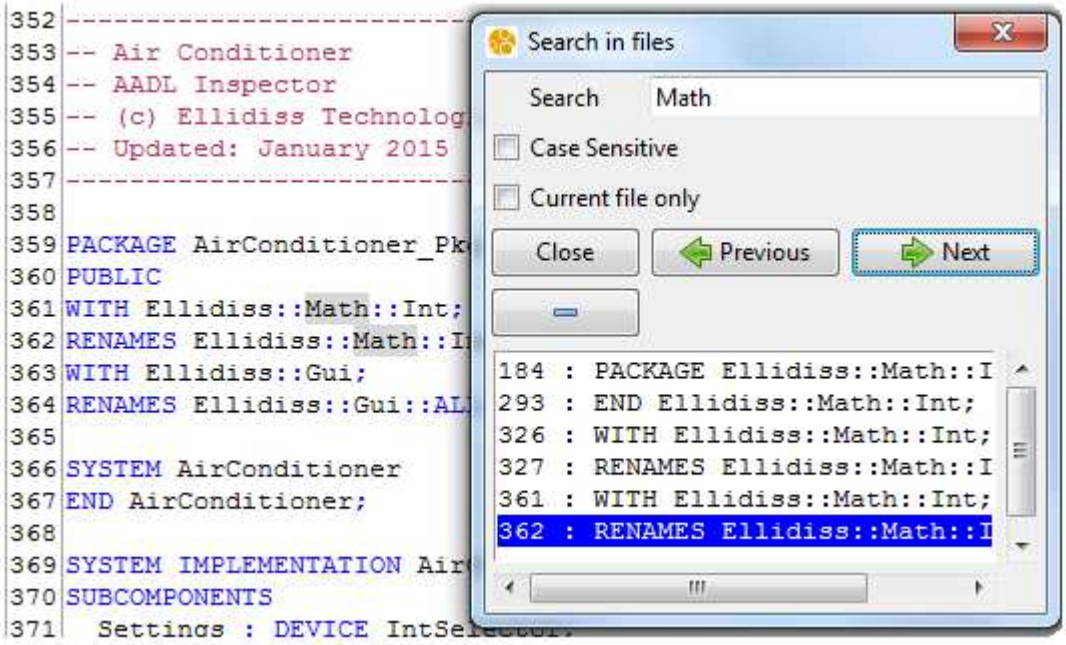
- two *navigation buttons*
- a set of *file selector tabs*
- a *file editing area*
- a *line number area*



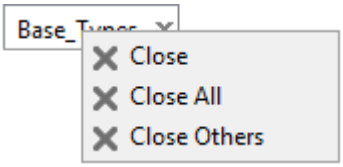
The *navigation buttons* provides access to all the loaded files, whereas the *file selector tabs* only show a limited selection of these files for quicker access.

To load a file in the editing area a drag and drop action is possible instead of using the *File/Load* menu: open the appropriate directory, select the desired file, depress and hold the left mouse button then drag the mouse until the **AADL Inspector** window is reached. Please note that this function is not available for Linux distributions.

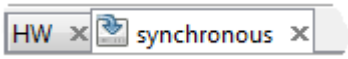
To find all the occurrences of a word in the displayed file, select the desired word and press the `Ctrl-F` key to open the search dialog box. Note that the *Next* button must be pressed to start the search.



A contextual menu (right mouse button click) is associated with the current *file selector tab*. It provides an easy way to close files. The effect of these menus is similar to the corresponding ones in the *File* menu. Note that clicking on the grey *cross* on the tab also closes the current file.



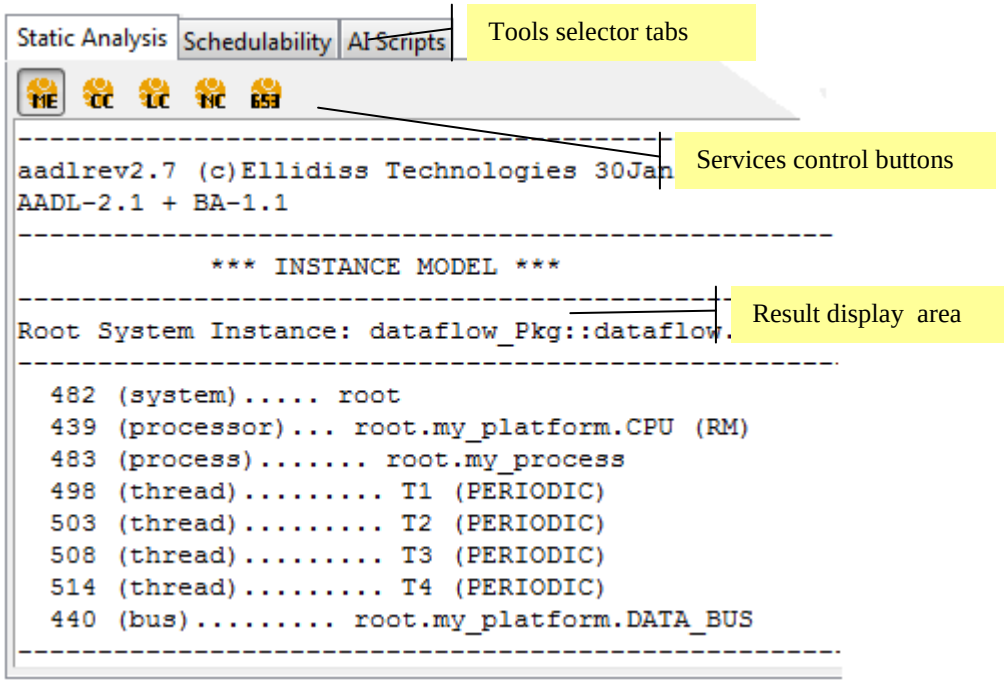
When a file has been modified, an icon appears on the tab to indicate that the changes have not been saved. Clicking on the *save* icon of the tab will save the file.



3.3 Processing tools area

The *processing tools area* allows the processing tool to be selected to apply changes on the **AADL** files loaded in the source files area, and display the corresponding execution result. This area is composed of:

- a set of *tool selector tabs*
- one or several *service control buttons*
- a read-only *result display area*



- *Tools selector tabs* can be configured by adding or removing tool description files (`.ais` files) in the `config` subdirectories of the installation directory

If one of the analysed files is modified, the background colour of the result display area becomes gray to indicate that the information is potentially out of date.

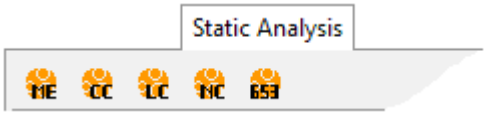
When the selected analysis tool can not be executed normally for the current **AADL** specification or if the **AADL** syntax is not correct, the corresponding error message will appear in the *Report* tab.

Clicking on the line number included in the error message will highlight the corresponding line in the *source files area*.

While working on large **AADL** projects, processing actions may take a significant time (up to a few minutes). Depending on the processing tool that is running, other user actions may not be allowed and the display may not be refreshed during that time.

3.3.1. Static Analysis

The static analysis tool encompasses a set of independent rules checkers that verify various facets of the semantic correctness of the source **AADL** specification. Each rules checker is implemented as a service of the static analysis tool and can be activated by pressing on the correspondent button:



- the *ME* button launches the Metrics checker.
- the *CC* button launches the **AADL** Consistency rules checker.
- the *LC* button launches the **AADL** Legality rules checker.
- the *NC* button launches the **AADL** Naming rules checker.
- the *653* button launches the **ARINC 653** rules checker.

More detailed explanations of the scope of each of these checkers can be found in separate user documents.

When an error, warning or information message is displayed by a processing tool, the line number of the corresponding **AADL** code is highlighted in both the *source files area* and *processing tools area*. Clicking on an error line number of the *processing tools area* automatically changes the source code display to make the erroneous line visible.

3.3.2. Schedulability analysis

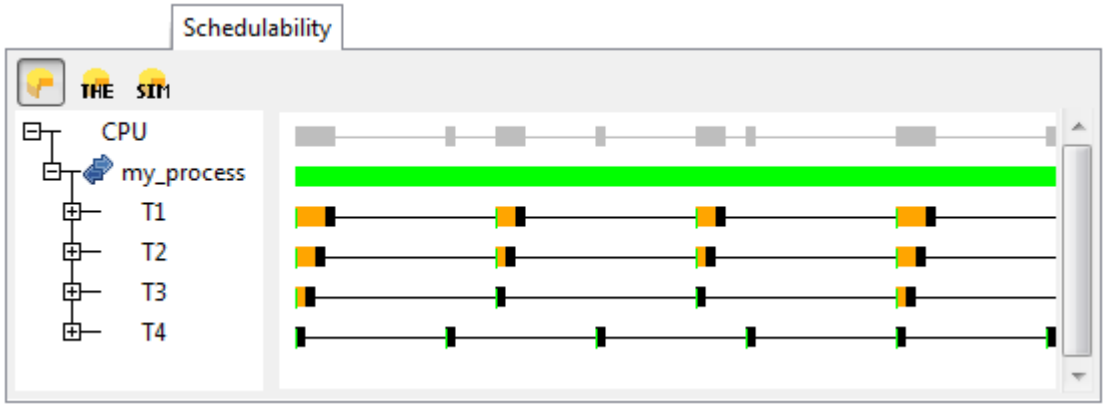
When the *schedulability* tab is selected, two buttons are presented to activate the analysis services provided by **Cheddar**:

- the first button computes the Schedule Table
- the *THE* button launches the Scheduling Theoretical Tests
- the *SIM* button launches the Scheduling Simulation Test

3.3.2.1. Schedule Table

Cheddar can produce a graphical representation of the timing behaviour of the real-time system being analysed. This *Schedule Table* is a result of the static simulation and may not be available on every kind of system.

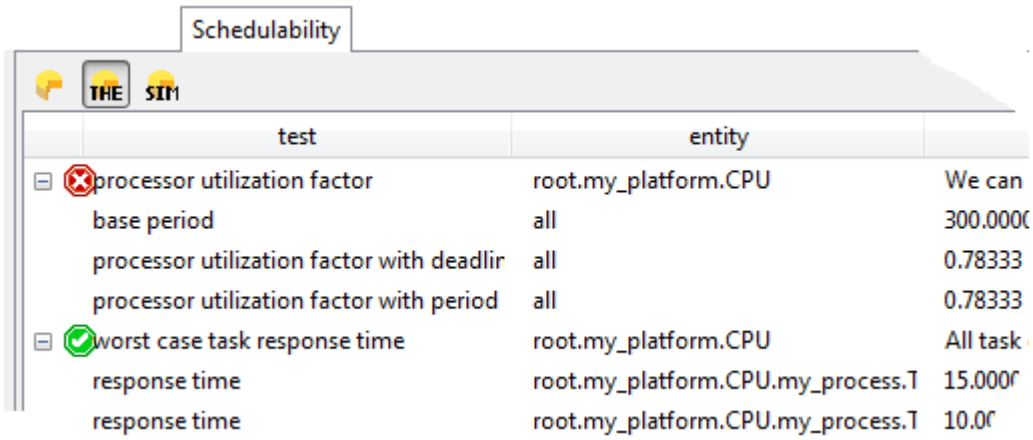
Note that the yellow button must be clicked to launch the computation of the *Schedule Table*.



Time lines are displayed for each Processor, Process, Thread and shared Data subcomponent in the current root System. The time scale and meaning of each used colour is shared with the dynamic simulator which is described below.

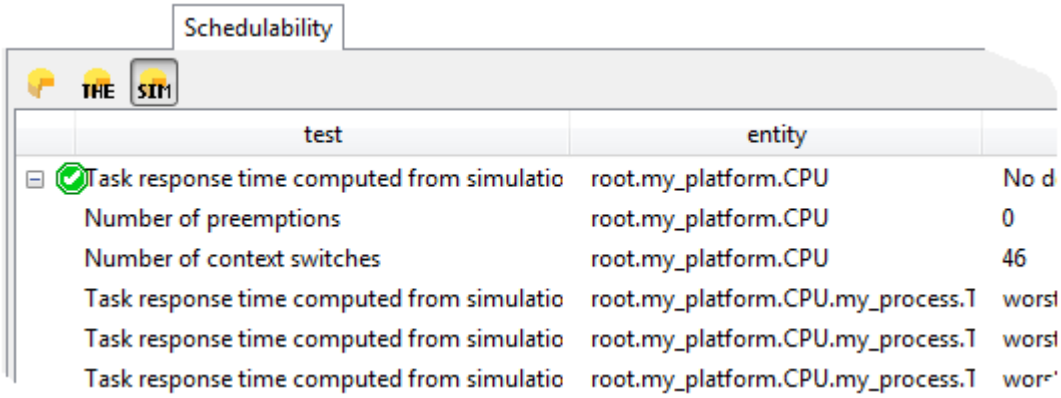
3.3.2.2. Scheduling Theoretical Tests

Theoretical tests compute the processor utilization factor and threads response time when the corresponding conditions are met.



3.3.2.3. Scheduling Simulation Tests

Simulation tests provide information about the number of pre-emption and context switches as well as threads response time. This static simulation can only be run for periodic systems.



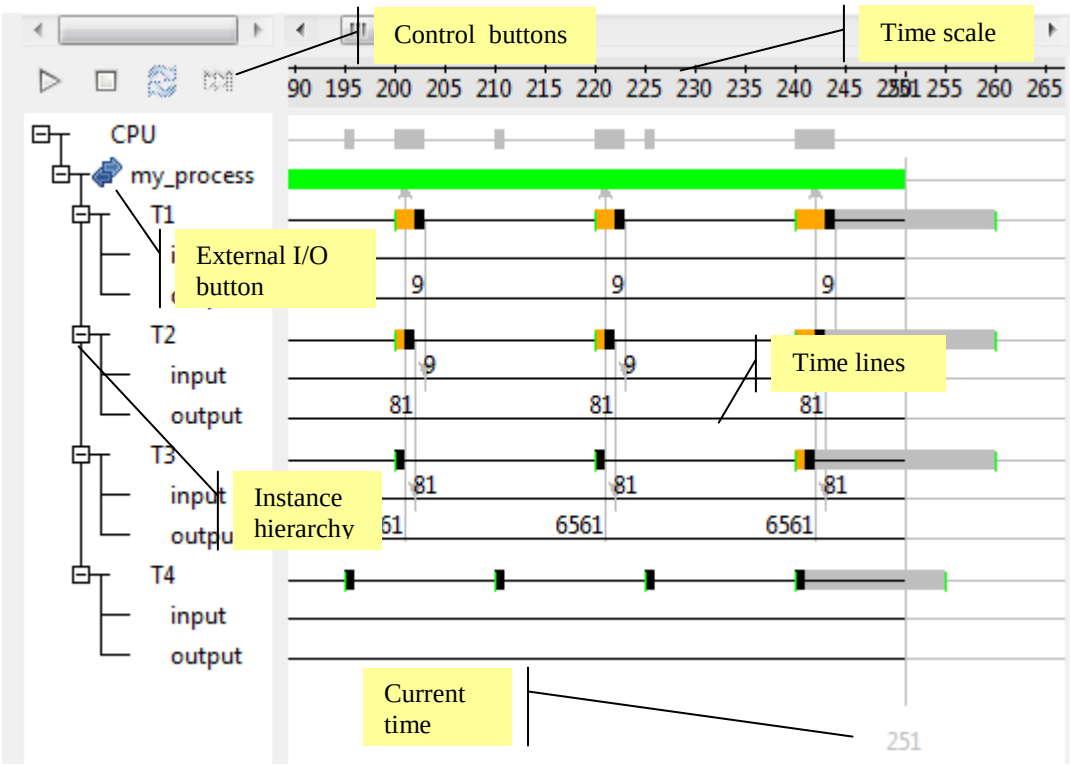
More detailed explanations about the scope of each of these tests can be found in a separate user document.

3.4 Simulation area

The *simulation area* is dedicated to controlling and displaying the output of the **Marzhin** dynamic simulator. This simulator complements the static simulator provided by **Cheddar** but is event-driven and can analyse a wider variety of real-time systems. The counter part is that the obtained time lines are not the result of mathematical computation and are thus less dependable.

The simulation area is composed of:

- a set of control buttons (same as in the Simulation Control Panel)
- a time scale (shared with the Cheddar Schedule Table)
- a deployable tree showing the AADL instance hierarchy
- an external I/O button on each process
- a simulator output area showing time lines for each Processor, Process, Thread and shared Data subcomponent in the current root System.



In addition, the *Simulation Control Panel* wizard can be used to set up the time scale and filter the entities displayed for the simulation. This wizard can be activated from the *Wizards* menu or the corresponding button in the main buttons bar. Refer to section 3.1.3.3 for more details.

3.4.1. Simulator action buttons

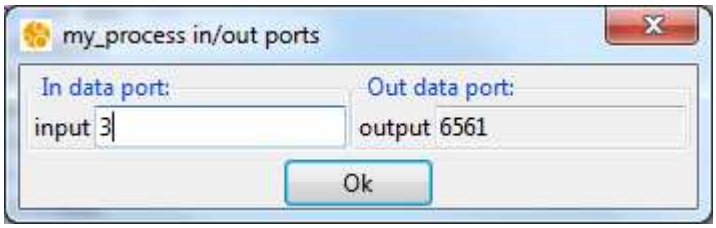
The simulator toolbar is composed of the following buttons:

	start the simulator
	pause the simulator
	stop the simulator
	open the simulator control panel
	refresh the simulation input

3.4.2. External I/O button

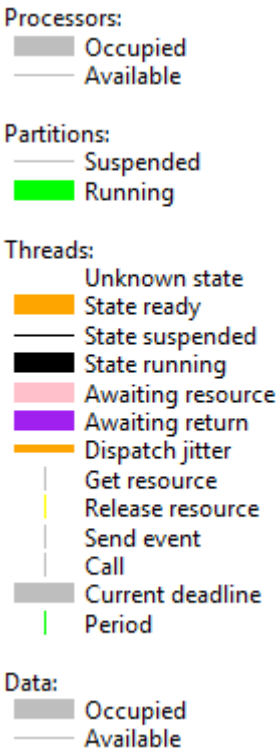
When a Process has ports that are connected downstream in the instance hierarchy, they can be displayed in a specific dialog box to allow the user to send in data and events and to show the result of out data and events. This dialog box can be opened by pressing the I/O button .Note that the value that is displayed for an out event port is the time of its last update.

```
489 PROCESS my_process
490 FEATURES
491   input : IN DATA PORT int;
492   output : OUT DATA PORT int;
493 END my_process;
```

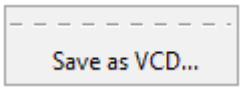


3.4.3. Simulation time lines

A separate time-line is shown for each Processor, each partition (Process), each Thread and each shared Data component. The colour code that is used for the time lines can be configured in the `AIConfig.ini` file and displayed in the *help* tab of the *Simulation control panel*. Default configuration is as follows:



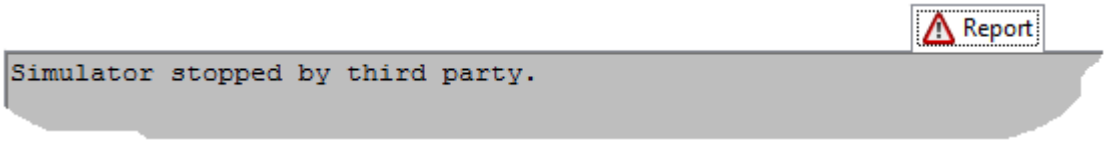
A contextual menu is available in the simulation area (as well as in the **Cheddar** *Schedule Table*) to save the current simulation trace into a file with the **VCD** (Value Change Dump) format.



3.5 Status bar and Error Report

The status bar located in the lower part of the window shows various informational or error messages generated by **AADL Inspector**:

When relevant, detailed error messages are displayed in the *Report* tab.



www.ellidiss.com

Sales office:
TNI Europe Limited
Triad House
Mountbatten Court
Worall Street
Congleton
Cheshire
CW12 1AG
UK
info@ellidiss.com
+44 1260 291 449

Technical support :
Ellidiss Technologies
24 quai de la douane
29200 Brest
Brittany
France

aadl@ellidiss.fr
+33 298 451 870